Computing exact geometric predicates using modular arithmetic with single precision

Hervé Brönnimann, Ioannis Z. Emiris, Victor Y. Y. Pan, Sylvain Pion

Appeared in 13th Annual ACM Symposium on Computational Geometry, June 4–6 1997, Nice, France.

Computing exact geometric predicates using modular arithmetic with single precision

Hervé Brönnimann* Ioannis Z. Emiris* Victor Y. Pan‡ Sylvain Pion*

Abstract: We propose an efficient method that determines the sign of a multivariate polynomial expression with integer coefficients. This is a central operation on which the robustness of many geometric algorithms depends. The method relies on modular computations, for which comparisons are usually thought to require multiprecision. Our novel technique of *recursive relaxation of the moduli* enables us to carry out sign determination and comparisons by using only floating point computations in single precision. The method is highly parallelizable and is the fastest of all known multiprecision methods from a complexity point of view. We show how to compute a few geometric predicates that reduce to matrix determinants. We discuss implementation efficiency, which can be enhanced by good arithmetic filters. We substantiate these claims by experimental results and comparisons to other existing approaches. This method can be used to generate robust and efficient implementations of geometric algorithms, including solid modeling, manufacturing and tolerancing, and numerical computer algebra (algebraic representation of curves and points, symbolic perturbation, Sturm sequences and multivariate resultants).

Keywords: computational geometry, exact arithmetic, robustness, modular computations, single precision, Residue Number Systems (RNS)

1 Introduction

Most of geometric predicates can be expressed as computing the sign of an algebraic expression. In principle, one may compute such expressions by using floating-point arithmetic with a fixed finite precision (f.p. arithmetic), but then the roundoff errors may easily lead to the wrong sign. This problem is often referred to as the *robustness* problem in computational geometry [17].

One solution to the robustness problem is to explicitly handle numerical inaccuracies, so as to design an algorithm that does not fail even if the numerical part of the computation is done approximately [19, 29], or to analyze the error due to the f.p. imprecision [12]. Such designs are extremely involved and have only been done for a few algorithms. The general solution, it has been widely argued, is to compute the predicates exactly [9, 6, 13, 33, 11]. This can be achieved in many ways: computing the algebraic expressions with infinite precision [31], with a finite but much higher precision that can be shown to suffice [14], or by using an algorithm that performs a specific test exactly. In the last category, much work has focused on computing the sign of the determinant of a matrix with integer entries [7, 2, 4], which applies to many geometric tests (such as orientation tests, in-circle tests, comparing segment intersections) as well as to algebraic primitives (such as resultants and algebraic representations of curves and surfaces). Recently, some techniques have been devised for handling arbitrary expressions and f.p. representation [28].

In our present paper, we propose a method that determines the sign of a multivariate polynomial expression with integer coefficients, using no operations other than modular arithmetic and f.p. computations with a fixed finite (single) precision. The latter operations can be performed very fast on usual computers. The Chinese remainder algorithms enable us to perform rational algebraic computations modulo several primes, that is, with a lower precision, and then to combine them together in order to recover the desired output value. The latter stage of combining the values modulo smaller primes, however, was always considered a bottleneck of this approach, because higher precision computations were required at this stage. Our paper proposes a new technique, which we call *recursive relax-*

*INRIA Sophia-Antipolis, B.P. 93, 2004, Route des Lucioles, 06902 Sophia-Antipolis Cedex, FRANCE. This research was partially supported by the ESPRIT IV LTR Project No. 21957 (CGAL).

‡ Department of Mathematics and Computer Science, Lehman College, City University of New-York, Bronx NY 10468, USA. Work on this paper was performed while visiting the second author and was supported by NSF Grants CCR 9020690 and 9625344 and PS-CUNY Awards 666327 and 667340.

ation of the moduli and which enables us to resolve the latter problem. Due to this technique, we correctly recover the sign of an integer from its value reduced modulo several smaller primes, and we only use some simple lower precision computations at the recovery stage. (This should make our algorithms of some independent interest also for the theory and practice of algebraic computing.) Our deterministic algorithms 1 and 2 of sections 3 and 4, respectively, specify our approach and our technique based on Lagrange's and Newton's interpolation formulae, respectively. Our algorithm 4 of section 5 gives a probabilistic simplification of algorithm 3. Preliminary experimental results and running times are discussed in section 7. In general, our methods are comparable in speed to other exact methods and even faster for particular inputs.

Related work. Performing exact arithmetic is usually expensive. Thus, it is customary to resort to arithmetic filters [14]: those filters safely evaluate a predicate in most cases, in order to avoid performing a more expensive exact implementation. The difficult cases arise when the expression whose sign we wish to compute is very small. For typical filters, the smaller this quantity, the slower the filter [7, 2, 28]: this is referred to as *adaptivity*. Modular arithmetic displays an opposite kind of adaptivity: with a smaller quantity, fewer moduli have to be computed, hence the test is faster. Typically, when filters fail, they also provide an upper bound on the absolute value of the expression whose sign we wish to compute (see many details and estimates in [25]). This bound can then be used to determine how many moduli should be taken. Modular arithmetic is therefore complementary to the filtering approach. We also observe this in section 7.

Residue Number Systems (RNS) express and manipulate numbers of arbitrary precision by their moduli with respect to a given set of numbers. They have been popular because they provide a cheap and highly parallelizable version of multiprecision arithmetic. It is impossible here to give a fair and full account on RNS, but Knuth [22] and Aho, Hopcroft, and Ullman [1] provide a good introduction to the topic. From a complexity point of view, RNS allows to add and multiply numbers in linear time. Its weak point is that sign computation and comparisons are not easily performed and seem to require full reconstruction in multiple precision, which defeats its purpose. This is precisely the issue that our paper handles.

The closest predecessors of our work are apparently [10] and [20]. The algorithm of Hung and Parhami [20] corresponds to single application of the second stage of our recursive relaxation of the moduli. Such a single application suffices in the context of the goal of [20], that is, application to divisions in RNS, but in terms of the sign determination of an integer, this only works for an absolutely larger input. The paper [10] gives probabilistic estimates for early termination of Newton's interpolation process, which we apply in our probabilistic analysis of our algorithm 4. Its main subject is an implementation of an algorithm computing multidimensional convex hulls. The paper [10] does not use our techniques of recursive relaxation of the moduli, and it does not contain the basic equations (1)–(3) of our section 3.

2 Exact sign computation using modular arithmetic

Modular computations. Our model of a computer is that of a f.p. processor that performs operations at unit cost by using b-bit precision (e.g., in the IEEE 754 double precision standard, we have $b = 53$). It is a realistic model as it covers the case of most workstations used in research and industry. We will use mainly one basic property of f.p. arithmetic on such a computer: for all four arithmetic operations (and for computing a square root too but we will not need it), the computed result is always the f.p. representation that best approximates the exact result [22, 28]. This means that the relative error incurred by an operation returning x is at most 2^{-b}, and that the absolute error is at most $2^{\lfloor \log |x| - b \rfloor}$. (All logarithms in this paper are base 2.) In particular, operations performed on pairs of integers smaller than 2^b are performed exactly as long as the result is also smaller than 2^b.

Let $m_1, \ldots, m_k$ be k pairwise relatively prime integers and let $m = \prod_i m_i$. For any number x (not necessarily an integer), we let $x_i = x \bmod m_i$ be the only number in the range $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ such that $x_i - x$ is a multiple of m_i. (This operation is always among the standard operations because it is needed for reducing the arguments of periodic functions.)

To be able to perform arithmetic modulo m_i on integers by using f.p. arithmetic with b-bit precision, we will assume that $m_i \leq 2^{b/2+1}$. Performing modular multiplications of two integers from the interval $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ can be done by multiplying these numbers and returning their product modulo m_i. (The product is smaller than 2^b in magnitude and hence is computed exactly.) Performing additions can be done very much in the same way, but since the result is in the range $\left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, taking the sum modulo m_i is more easily achieved by adding or subtracting m_i if necessary. Integral divisions modulo m_i can be computed using Euclid's algorithm; we will need them in this paper only in section 6. Therefore, arithmetic modulo m_i can be performed using f.p. arithmetic with b-bit precision, provided that $m_i \leq 2^{b/2+1}$.

Exact sign computation. In this paper, we consider the following computational problem.

Problem 1 *Let k, b, $m_1, \ldots, m_k$ denote positive integers, $m_1, \ldots, m_k$ being pairwise relatively prime, such that $m_i \leq 2^{b/2+1}$, and let $m = \prod_{i=1}^{k} m_i$. Let x be an integer whose magnitude is smaller than $\lfloor (m/2)(1 - k2^{-b}) \rfloor$. Given $x_i = x \bmod m_i$, compute the sign of x by using only modular and floating-point arithmetic both performed with b-bit precision.*

We will solve this problem, even though x can be huge and, therefore, not even representable by using b bits.

3 Lagrange's method

According to the Chinese remainder theorem [8], x is uniquely determined by its residues x_i, that is, Problem 1 is well defined and admits a unique solution. Moreover, this solution can be derived algorithmically from the following formula, due to Lagrange [22, 23]. If x is an integer in the range $\left[-\frac{m}{2}, \frac{m}{2}\right)$, x_i stands for the residue $x \bmod m_i$, $v_i = m/m_i = \prod_{j \neq i} m_j$, and $w_i = v_i^{-1} \bmod m_i$, then

$$x = \left(\sum_{i=1}^{k} ((x_i w_i) \bmod m_i)\, v_i \right) \bmod m.$$

Trying to determine the sign of such an integer, we computed the latter sum approximately in fixed b-bit precision. Computing a linear combination of large integers v_i with its subsequent reduction modulo m can be difficult, so we prefer to compute the number

$$S = \mathrm{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i) \bmod m_i}{m_i} \right),$$

where $\mathrm{frac}(z)$ is the fractional part of a number x that belongs to $\left[-\frac{1}{2}, \frac{1}{2}\right)$.

If S were computed exactly, then we would have $S = \mathrm{frac}(x/m)$, due to Lagrange's interpolation formula. In fact, S is computed with a fixed b-bit precision. Nevertheless, we prove in the full version that exact rounding and summing terms pairwise in a tree-like fashion introduces an absolute error $\varepsilon_k = k\, 2^{-b-1}$ in the sum S. Therefore, if S is greater than ε_k, the sign of x is the same as the sign of S, and we are done. Otherwise, $|x| < \varepsilon_k m$. Since $m_k \leq 2^{b/2+1}$, we can say conservatively that for all practical values of k and b, this is smaller than $\frac{m}{2m_k}(1 - \varepsilon_{k-1})$, and hence we may recover x already from $x_i = x \bmod m_i$ for $i = 1, \ldots, k-1$, that is, it suffices to repeat the computation using only $k-1$, rather than k moduli. Recursively, we will reduce the solution to the case of a single modulus m_1 where $x = x_1$. We will call this technique *recursive relaxation of the moduli*, and we will also apply it in section 4.

We will present our resulting algorithm by using additional notation:

$$\begin{aligned}
m^{(j)} &= \prod_{1 \leq i \leq j} m_i, \\
v_i^{(j)} &= \prod_{\substack{1 \leq i \leq j \\ i \neq j}} m_i, \\
w_i^{(j)} &= \left(v_i^{(j)} \right)^{-1} \bmod m_i, \\
S^{(j)} &= \mathrm{frac}\left(\sum_{i=1}^{j} \frac{x_i w_i^{(j)} \bmod m_i}{m_i} \right),
\end{aligned}$$

so that $m = m^{(k)}$, $w_i = w_i^{(k)}$ and $S = S^{(k)}$. All the computations in this algorithm are performed by using f.p. arithmetic with b-bit precision.

Algorithm 1 : Compute the sign of x knowing $x_i = x \bmod m_i$

Precomputed data: *m_j, $w_i^{(j)}$, ε_j, for all $1 \leq i \leq j \leq k$*
Input: *integers k and $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$, for all $1 \leq i \leq k$*
Output: *sign of x, the unique solution of $x_i = x \bmod m_i$ in $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$*
Precondition: $|x| \leq \frac{m^{(k)}}{2}(1 - \varepsilon_k)$

1. *Let $j \leftarrow k + 1$*
2. *Repeat $j \leftarrow j - 1$,*
   $S^{(j)} \leftarrow frac\left(\sum_{i=1}^{j} \frac{x_i w_i^{(j)} \bmod m_i}{m_i} \right)$
   until $|S^{(j)}| > \varepsilon_j$ or $j = 0$
3. *If $j = 0$ return "$x = 0$"*
4. *If $S^{(j)} > 0$ return "$x > 0$"*
5. *If $S^{(j)} < 0$ return "$x < 0$"*

Lemma 3.1 *Algorithm 1 computes the sign of x knowing its residues x_i by using at most $\frac{k(k-1)}{2}$ f.p. modular multiplications, $\frac{k(k-1)}{2}$ f.p. divisions, $\frac{k(k-1)}{2}$ f.p. additions, and $k + 2$ f.p. comparisons.*

Proof. The m_i's and the $w_i^{(j)}$'s are computed once and for all and placed into a table, so they are assumed to be available to the algorithm at no cost. In step 2, a total of j modular multiplications, j f.p. divisions, and j f.p. additions (including taking the fractional part) are performed. □

By using parallel implementation of the summation of k numbers on $\lceil k/\log k \rceil$ arithmetic processors in $2\lceil \log k \rceil$ time (cf. e.g. [3, ch.4]), we may perform algorithm 1 on $\lceil k/\log k \rceil$ arithmetic processors in $O(k \log k)$ time, assuming each b-bit f.p. operation takes constant time. Furthermore, if $\lceil k^2/\log k \rceil$ processors are available, we may compute all the $S^{(j)}$ and compare $|S^{(j)}|$

with ε_j, for all $j = 1, \ldots, k$ concurrently. This would require $O(\log k)$ time on $\lceil k^2/\log k \rceil$ processors. Finally, if $\lceil tk/\log k \rceil$ processors are available for some parameter $1 \le t \le k$, we may perform algorithm 1 in $O((k \log k)/t)$ time by batching $\lceil t \rceil$ consecutive values of j in parallel. In practice, the algorithm needs to examine only a few values of j, so $O(\log k)$ time suffices even with $\lceil k/\log k \rceil$ processors.

Remark 1. If actually $x = 0$, the algorithm can be greatly sped up by testing if $x_j = 0$ in step 2, in which case we may directly pass to $j - 1$. Furthermore, stage 3 is not needed unless $x = x_j = 0$ for all j, which can be tested beforehand. Of course, if the only answer needed is "$x = 0$" or "$x \neq 0$", then it suffices to test if all the x_i's are zero.

Remark 2. If $|x|$ is not too small compared to $m^{(k)}$, then only step k is performed, involving only k f.p. operations of each kind. Also, we note that the costly part of the computation is likely to be the determination of the x_i's. For these reasons, we should try to minimize the number k of moduli m_i involved in the algorithm. This can be done by getting better upper estimates on the magnitude of the output or by using the probabilistic technique of section 5.

4 A generalization of Lagrange's method

We will show that Lagrange's method is in fact a particular case of the following method. Let

$$\Sigma^{(0)} = S^{(k)} = \text{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i) \bmod m_i}{m_i}\right).$$

This quantity is computed in the first step of algorithm 1. If the computed value of $\Sigma^{(0)}$ is smaller than ε_k, it implies that $\Sigma^{(0)} < 2\varepsilon_k$. Thus, $|x|$ is smaller than $2m\varepsilon_k$. We can then multiply $x_i w_i$ by

$$\alpha_k = \lfloor \frac{\frac{1}{2}(1 - \varepsilon_k)}{2\varepsilon_k} \rfloor,$$

to obtain $(x_i w_i \alpha_k) \bmod m_i$ for all $i = 1, \ldots, k$. This can be easily done by precomputing α_k modulo each m_i. We then compute

$$\Sigma^{(1)} = \text{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i \alpha_k) \bmod m_i}{m_i}\right),$$

and more generally,

$$\Sigma^{(j)} = \text{frac}\left(\sum_{i=1}^{k} \frac{(x_i w_i \alpha_k^j) \bmod m_i}{m_i}\right),$$

where we assume $\alpha_k \bmod m_i$ precomputed for all $i = 1, \ldots, k$. This leads to the following algorithm:

Algorithm 2 : Generalized Lagrange's method. Compute the sign of x knowing $x_i = x \bmod m_i$.

Precomputed data: m_i, w_i, ε_k, $\alpha_k \bmod m_i$, *for all* $i = 1, \ldots, k$
Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for all* $i = 1, \ldots, k$
Output: *sign of* x, *the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m}{2}, \frac{m}{2}\right)$
Preconditions: $|x| \le \frac{m}{2}(1 - \varepsilon_k)$ *and* $x \neq 0$

1. *Let* $j \leftarrow -1$
2. *Repeat* $j \leftarrow j + 1$,
   $\Sigma^{(j)} \leftarrow frac\left(\sum_{i=1}^{k} \frac{x_i w_i \bmod m_i}{m_i}\right)$
   $x_i \leftarrow x_i \alpha_k \bmod m_i$ *for all* $1 \le i \le k$,
   until $|\Sigma^{(j)}| > \varepsilon_k$ *or* $j = k$
3. *If* $j = k$ *return "*$x = 0$*"*
4. *If* $\Sigma^{(j)} > 0$ *return "*$x > 0$*"*
5. *If* $\Sigma^{(j)} < 0$ *return "*$x < 0$*"*

It is easy to see that the number of iterations in step 2 is $O(\log m/\log \alpha_k) = O(\log k)$, because $|x|$ is no less than 1 and no more than $m^{(k)} \le 2^{k(b/2+1)}$, and is multiplied by α_k at each iteration.

Remark 3. Algorithm 1 corresponds to a choice of $\alpha_k = m_j$ in step j, this is why we call algorithm 2 a generalization. Applying Lagrange's method, we do not multiply by the maximum at each step, but by a smaller number chosen so as to simplify the computations.

Remark 4. To yield the parallel time bounds such as $O(\log k)$ using $\lceil k^2/\log k \rceil$ processors for algorithm 2, we need to precompute ${\alpha_k}^j$ for all $i, j = 1, \ldots, k$.

5 An incremental variant

A recursive incremental version of the Chinese remainder algorithm, named after Newton, is described in this section. Its main advantage is that it does not require an *a priori* bound on the magnitude of x.

Let $x^{(j)} = x \bmod m^{(j)}$, for $j = 1, \ldots, k$, so that $x^{(1)} = x_1$ and $x = x^{(k)}$. Let $y_1 = x_1$, and for all $j = 2, \ldots, k$,

$$\begin{aligned} t_j &= w_j^{(j)} = (m^{(j-1)})^{-1} \bmod m_j, \\ y_j &= \left(x_j - x^{(j-1)}\right) t_j \bmod m_j \in \left[-\frac{m_j}{2}, \frac{m_j}{2}\right). \end{aligned}$$

Then (see, e.g., [22, 23]), for all $j = 2, \ldots, k$,

$$x^{(j)} = x^{(j-1)} + y_j m^{(j-1)}.$$

Clearly, this leads to an incremental computation of the solution $x = x^{(k)}$ to problem 1; we see below how this

can be exploited for an early termination of the interpolation. A further advantage is that all computation can be kept modulo m_j, and no floating-point computation is required, in contrast to sections 3 and 4 where $S^{(j)}$ or $\Sigma^{(j)}$ are computed. It is obvious, that when $y_j \neq 0$, then the sign of $x^{(j)}$ is the same as the sign of y_j since $|x^{(j-1)}| \leq m^{(j-1)}/2$. If $y_j = 0$, the sign of $x^{(j)}$ is the same as that of $x^{(j-1)}$, for $j \geq 2$, whereas the sign of $x^{(1)} = x_1 = y_1$ is known. If $y_j = 0$ for all j, then this is precisely the case when $x = 0$.

For $1 \leq i < j \leq k$, we introduce integers

$$u_i^{(j-1)} = \left(m^{(i-1)} t_j\right) \bmod m_j = \left(\prod_{l=i}^{j-1} m_l\right)^{-1} \bmod m_j.$$

Then $t_j = u_1^{(j-1)}$. In the full version of the paper, we show that the quantities y_j verify the following equality for all $j = 2, \ldots, k$,

$$y_j = \left((x_j - x_1) u_1^{(j-1)} - \sum_{i=2}^{j-1} y_i u_i^{(j-1)}\right) \bmod m_j.$$

Therefore, they can be computed by using modular arithmetic with bit-precision given by the maximum bit-size of the m_j^2. Here it suffices to assume that the absolute value of x is bounded by $m^{(k)}/2$.

Algorithm 3 : Compute the sign of x, knowing $x \bmod m_i$, by Newton's incremental method

Precomputed data: m_j, $u_i^{(j-1)}$, *for all* $1 \leq i < j \leq k$

Input: *integers* k *and* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for all* $i = 1, \ldots, k$

Output: *sign of* x, *where* x *is the unique solution of* $x_i = x \bmod m_i$ *in* $\left[-\frac{m^{(k)}}{2}, \frac{m^{(k)}}{2}\right)$

Precondition: *None.*

1. *Let* $y_1 \leftarrow x_1$, $j \leftarrow 1$. *Depending on whether* y_1 *is negative, zero or positive, set* s *to* $-1, 0$ *or* 1, *respectively.*
2. *Repeat* $j \leftarrow j + 1$,

$$y_j \leftarrow \left((x_j - x_1) u_1^{(j-1)} - \sum_{i=2}^{j-1} y_i u_i^{(j-1)}\right) \bmod m_j,$$

until $j = k$. *For every* j, *if* y_j *is strictly negative or positive, then set* s *to* -1 *or* 1, *respectively.*

3. *Depending on whether* s *is* $-1, 0$, *or* 1, *return "*$x < 0$,*" "*$x = 0$,*" or "*$x > 0$*", respectively.*

Remark 5. As in remark 1, we can test beforehand if all $x_i = 0$, which is precisely the case when $x = 0$.

Lemma 5.1 *Algorithm 3 computes the sign of* x *knowing its residues* x_i *using at most* $\frac{k(k-1)}{2}$ *f.p. modular multiplications,* $\frac{k(k-1)}{2}$ *f.p. modular additions, and* k *f.p. comparisons.*

Proof. For every $j = 2, \ldots, k$, there are $j - 1$ f.p. modular additions and multiplications. There is one comparison for each $j = 1, \ldots, k$. □

Algorithm 3 requires k recursive steps in the worst case (though we expect to have it terminate earlier in practice), so its parallel time cannot be decreased below $\Omega(k \log k)$. Nevertheless the algorithm can be implemented in $O(k \log k)$ time on $\lceil k/\log k \rceil$ processors, assuming each b-bit f.p. operation takes constant time.

To compare with algorithm 1, realistically assume that a modular addition is equivalent to $3/2$ f.p. additions and one comparison, on the average. Then, algorithm 1 requires $\frac{k(k-1)}{2}$ f.p. divisions (which are essentially multiplications with precomputed reciprocals) more than algorithm 3, whereas the latter requires $\frac{k(k-1)}{4}$ extra f.p. additions and $\frac{k(k-1)}{2}$ additional comparisons.

The principal feature of this approach, based on Newton's formula for recovering x, is its incremental nature. This may lead to faster termination, before examining all k moduli. Informally, this should happen whenever the magnitude of x is significantly smaller than $m^{(k)}/2$, in which case we would save the computation required to obtain x_j for all larger j. This saves a significant amount of computation if termination occurs earlier than the static bound indicated by k. A quantification of this property in the case of convex hulls can be found in [10].

We propose below a probabilistic variant of algorithm 3 which, moreover, removes the need of an *a priori* knowledge of k. Step 2 is modified to include a test of y_j against zero. Clearly, $y_j = 0$ precisely when $x^{(j)} = x^{(j-1)}$. Then we may deduce that $x^{(j)} = x^{(k)} = x$, with a very high probability, and terminate the iteration.

Algorithm 4 : Yield earlier termination of algorithm 3 for absolutely smaller input. Algorithm 3 is modified exactly as shown.

Input: *integers* $x_i \in \left[-\frac{m_i}{2}, \frac{m_i}{2}\right)$ *for* $i = 1, \ldots$ *as required in the course of the algorithm; no need for* k

Output: *sign of* x *with very high probability*

2. *Terminate the loop also if* $y_j = 0$

By lemma 3.1 of [10], this algorithm fails with probability bounded by $(k-2)/m_{\min}$, where

$$m_{\min} = \min\{m_1, m_2, \ldots, m_k\}.$$

For $k \leq 12$, $m_{\min} \geq 2^{25}$, the error probability is less than 10^{-6}. A more careful analysis can reduce this probability by exploiting the correlation of failure at different stages. For experimental support of this claim, we refer to [10].

6 Applications

6.1 Exact geometric predicates

Exact geometric predicates is the most general way to provide robust implementations of geometric algorithms [9, 13, 33, 11]. In particular, orientation tests can be implemented by looking for the sign of a determinant. Modular arithmetic becomes increasingly interesting when the geometric tests (e.g. determinants) are of higher order and complexity. They are central in, notably:

- Convex hull computations: this is a fundamental problem of computational geometry and of optimization for larger dimensions. Among numerous practical applications, one may note collision detection in dynamic simulation and animation [24], prediction of poses for industrial parts on a conveyer belt, and computation of stable grasps by robots [30].
- Voronoi diagrams: for points, their computation reduces to convex hulls. The sweepline algorithm in 2D is relatively simple, but involves tests of degree 20 and modular arithmetic can be of substantial help in conjunction with arithmetic filters [14]. For segments, the tests become of even higher degree and complexity [6], and f.p. computation is likely to introduce errors, so exact arithmetic is often a must.
- Mixed subdivisions used in solving systems of nonlinear equations. Sparse elimination theory is a relatively new area of computational algebraic geometry, which exploits the geometric structure of polynomial systems in order to obtain tighter bounds and faster algorithms for their manipulation [15]. The algebraic questions are formulated in terms of *Newton polytopes* and their mixed volume, each polytope being the convex hull of the exponent vectors appearing in a polynomial.

Even for small dimensions, the nature of the data may force the f.p. computation to introduce inconsistencies, for instance, in:

- Planarity testing in geometric tolerancing [32]. Here, one must determine if a set of points sampling a plane surface can be enclosed in a slab whose width is part of the planarity requirements. The computation usually goes by computing the width of the convex hull, and the data is usually very flat, hence prone to numerical inaccuracies.

In geometric and solid modeling, traditional approaches have employed finite precision floating point arithmetic, based on bounds on the roundoff errors. Although certain basic questions in this domain are now considered closed, there remain some fundamental open problems, including boundary computation [18]. Tolerance techniques and symbolic reasoning have been used, but have been mostly restricted to polyhedral objects; their extension to curved or arbitrary degree sculptured solids would be complicated and expensive. More recently, exact arithmetic has been proposed as a valid alternative for generating boundary representations of sculptured solids, since it guarantees robustness and precision even for degenerate inputs at a reasonable or negligible performance penalty [21].

Furthermore, exact arithmetic allows the use of a variety of algebraic and symbolic methods, including algebraic representation of curves and points, symbolic perturbation, Sturm sequences and multivariate resultants; for an introduction to these methods, see [5]. The critical operation is deciding the sign of a multivariate polynomial expression with rational coefficients on a set of points. Recent data structures that exploit structure of algebraic objects, such as *straight-line programs*, also use precisely this kind of primitive operation [27].

6.2 Sign of the determinant of a matrix

As mentioned, computing the sign of a matrix determinant is a basic operation in computational geometry, applied to many geometric tests (such as orientation tests, in-circle tests, comparing segment intersections) [7, 2]. Sometimes, the entries to the determinant are themselves algebraic expressions. For instance, the in-circle test can be reduced to computing a 2×2 determinant, whose entries have degree 2 and thus require $2b + O(1)$-bit precision to be computed exactly [2]. Computing these entries by using modular arithmetic enables in-circle tests with b-bit precision while still computing exactly the sign of a 2×2 determinant.

To compute an $n \times n$ determinant modulo m_k, we may use Gaussian elimination with a single final division. At step $i < n$ of the algorithm, the matrix is

$$\begin{pmatrix} \vdots & \dots & \dots \\ 0 & \alpha_{i,i} & \dots \\ \vdots & \vdots & \vdots \\ 0 & \alpha_{n,i} & \dots \end{pmatrix}$$

and we assume that the pivot $\alpha_{i,i}$ is invertible modulo m_k. Then we change line L_j to $\alpha_{i,i}L_j - \alpha_{j,i}L_i$ for all $j = i+1, \dots, n$. At step n of the algorithm, we multiply the coefficient $\alpha_{n,n}$ by the modular inverse of the product $\prod_{i=1}^{n-1} \alpha_{i,i}^{n-i}$. This gives us the value of the determinant modulo m_k. Note that the same method but with non-modular integers and a final division would have involved exponentially large integers and several

slow divisions at each step. Nevertheless, it is only the range of the final result that matters for modular computations. This shows a big advantage of modular arithmetic over other multiprecision approaches.

The pivots should be invertible modulo m_k. If m_k is prime, the pivot simply has to be non-zero modulo m_k. The algorithm can be also easily implemented if m_k is a power of a prime, or if m_k is the product of two primes. This would be desirable mainly for taking $m_k = 2^{b_k}$ for which modular arithmetic is done naturally by integer processors, though here, special care must be taken about even output. Other choices of m_k do not seem to bring any improvement.

With IEEE double precision ($b = 53$), we choose moduli smaller than 2^{27}, so that $2(\frac{m_k}{2})^2 \leq 2^{53}$: Gaussian elimination intensively uses $(ad - bc)$-style operations; here we may apply one final modular reduction, instead of two for each product before subtracting.

This algorithm performs $O(n^3)$ operations for each modulus m_i. With Hadamard's determinant bound and m_k greater than $2^{b/2}$, only $k = \lceil 2n \log n \rceil$ finite fields need to be considered. Hence the complexity of finding the sign of the determinant is $O(n^4 \log n)$ single precision operations. On a $O(n^3 \log n)$-processor machine, the time complexity drops to $O(n)$, if we use custormary parallelization of the Gaussian elimination routine for matrix triangulation (cf. [16]), which gives us the value of the determinant. (We apply this routine in modular arithmetic, with simplified pivoting, concurrently for all m_i's.) Theoretically, substantial additional parallel acceleration can be achieved by using randomization [3, ch. 4], [26], yielding the time bound $O(\log^2 n)$ on $\lceil n^3 \log n \rceil$ arithmetic processors, and the processor bound can be decreased further to $O(n^{2.376})$, by applying asymptotically fast algorithms for matrix multiplication.

7 Experimental results

We present several benchmark results of our implementations of the described methods for computing a determinant in C, and compare them with different existing packages. Method FP is a straightforward f.p. implementation of Gaussian elimination. Method LEDA uses the routine `sign_of_determinant(integer_matrix)` of Leda [6]. Method CL has been implemented by us based on [7, 4]. As we compare with methods that handle arbitrary dimensions, we did not specialize the implementation for small dimensions as is done in [4] (this would provide an additional speedup of approximately 3). Method GMP is an implementation of Gaussian elimination using the GNU Multiprecision Package, for dimension lower than 5, and an implementation of Bareiss' extension of Gaussian elimination, for higher dimensions. Method MOD is an implementation of

n	FP	MOD	CL	GMP	LEDA
2	3.5	28	47	12	243
3	9	71	131	61	1032
4	18	151	276	226	2847
5	32	525	503	660	6370
6	55	857	850	1820	12330
7	85	1410	1213	3810	23400
8	130	2150	1954	6700	38700
9	182	3370	2669	11080	60500
10	250	4810	3671	17100	89000
11	330	6340	5063	27200	135800
12	420	8410	6559	39100	191900
13	550	11600	9272	53000	210750
14	700	14930	11461	72400	255750

Table 1: Performance on random determinants.

n	FP	MOD	CL	GMP	LEDA
2	3.5	30	344	12	240
3	9	75	832	60	1025
4	18	155	1472	220	2605
5	32	532	2598	655	5880
6	55	871	4419	1810	11370
7	85	1420	7040	3800	20700
8	130	2160	9590	6690	38300
9	182	3390	13610	11060	60300
10	250	4840	18590	17080	94000
11	330	6360	24850	27170	130000
12	420	8450	33800	39050	195000
13	550	11670	42325	52940	205700
14	700	15110	52800	72330	240500

Table 2: Performance on small determinants.

n	FP	MOD	CL	GMP	LEDA
2	3.5	31	343	12	230
3	9	76	912	57	890
4	18	157	2195	220	2500
5	32	540	4647	650	5820
6	55	885	8270	1800	12290
7	85	1440	13790	3790	20780
8	130	2180	21110	6670	34420
9	182	3420	31610	11040	55090
10	250	4870	44800	17050	92640
11	330	6400	62920	27150	136520
12	420	8480	87300	39020	184000
13	550	11700	119800	52920	201500
14	700	15150	143650	72250	248000

Table 3: Performance on zero determinants.

modular Gaussian elimination as described in section 6 using our recursive relaxation of the moduli. Of the other methods available, the lattice method of [4] has not yet been implemented in dimensions higher than 5; LN [14] provides a very fast implementation in dimensions up to 5 but was not available to us in higher dimensions.

Among the methods that guarantee exact computation, our implementations are at least as efficient as the others, and for certain classes of input they outperform all available programs. Furthermore our approach applies to arbitrary dimensions, whereas methods that compute a f.p. approximation of the determinant value are doomed to fail in dimensions higher than 15 because of overflow in the f.p. exponent.

All tests were carried out on a 85MHz Sun Sparc 5 workstation, using the `clock()` function. Each program is compiled with the compiler that gives best results. Each entry in the following tables represents the average time of one run in microseconds, with a maximum deviation of about 10%. We concentrated on determinant sign evaluation and considered three classes of matrices: random matrices, whose determinant is typically away from zero, in table 1, almost-singular matrices with single-precision determinant in table 2, and lastly singular matrices with zero determinant in table 3. The coefficients are integers of bit-size $53 - n$ (due to restrictions of Clarkson's method).

Our results suggest that our approach is comparable, and for certain classes of input significantly faster than the examined alternatives that guarantee exact results. The running times are displayed in tables 1–3. (For small dimensions, specialized implementations can provide an additional speedup for all methods.) Our code is reasonably compact and easy to maintain. As an obvious improvement, with a reasonably accurate f.p. filter, the penalty of exact arithmetic can be paid only for small determinants (tables 2 and 3). Another improvement we plan on exploring is to use parallelization.

Some side effects may occur, due to the way we generate matrices. The code of the modular package is free, and anyone can benchmark it on the kind of matrices that he uses. It is available via the URL `http://www.inria.fr/prisme/personnel/pion/progs/modular.html`

8 Conclusion

RNS systems have been used in number systems because they provide a highly parallelizable technique for multiprecision. As parallel computers are becoming more available, RNS provide an increasingly desirable implementation of multiprecision. This comes in sharp contrast with other multiprecision methods that are not easily parallelizable. Perhaps the main problem with RNS is that comparisons and sign computations seem to require full reconstruction and, therefore, use standard multiprecision arithmetic. We show that one may in fact use only single precision and still perform these operations exactly and efficiently. In some applications, the number of moduli may be large. Our algorithms may be easily implemented in parallel with a speedup depending almost linearly on the number of processors.

As an application, we show how to compute the sign of a determinant. This problem has received considerable attention, yet the fastest techniques are usually iterative and do not parallelize easily. Moreover, they usually only handle single precision inputs. Our techniques are comparable in speed or even faster than other techniques (e.g. [2, 6, 7]), and can easily handle arbitrarily large inputs.

A central problem we plan to explore further is to design algorithms that compute upper bounds on the quantities involved to determine how many moduli should be taken. For determinants, the static bounds we use seem to suffice for applications in computational geometry [14]. They might be overly pessimistic in other areas (such as tolerancing or symbolic algebra) where the nature of the data or algebraic techniques might imply much better bounds. A valid approach we will further study and implement is Newton's incremental method of section 5, provided that we are willing to afford some small probability of error.

References

[1] A. V. Aho and J. E. Hopcroft and J. D. Ullman. *The Design And Analysis Of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. Preparata, and M. Yvinec. Evaluation of a new method to compute signs of determinants. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C16–C17, 1995.

[3] D. Bini, V. Y. Pan. *Polynomial and Matrix Computations. Vol. 1: Fundamental Algorithms*. Birkhäuser, Boston, 1994.

[4] H. Brönnimann, M. Yvinec. Efficient exact evaluation of signs of determinants. These proceedings.

[5] B. Buchberger, G.E. Collins, and R. Loos, editors. *Computer Algebra: Symbolic and Algebraic Computation*. Springer, Wien, 2nd edition, 1982.

[6] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995. Package available at `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[7] K. L. Clarkson. Safe and effective determinant eval-

uation. In *Proc. 33rd Annu. IEEE Sympos. Found. Comput. Sci.*, pages 387–395, 1992.

[8] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Mass., 1990.

[9] H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graphics*, 9(1):67–104, 1990.

[10] I.Z. Emiris. A complete implementation for computing general dimensional convex hulls. *Intern. J. Computational Geom. & Applications*, 1997. To appear. Preliminary version as Tech. Report 2551, INRIA Sophia-Antipolis, France, 1995.

[11] I.Z. Emiris and J.F. Canny. A general approach to removing degeneracies. *SIAM J. Computing*, 24(3):650–664, 1995.

[12] S. Fortune. Numerical stability of algorithms for 2-d Delaunay triangulations and Voronoi diagrams. In *Proc. 8th Annu. ACM Sympos. Comput. Geom.*, pages 83–92, 1992.

[13] S. Fortune and C. J. Van Wyk. Efficient exact arithmetic for computational geometry. In *Proc. 9th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1993.

[14] S. Fortune, C.J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Trans. on Graphics*, 1996.

[15] I.M. Gelfand, M.M. Kapranov, and A.V. Zelevinsky. *Discriminants and Resultants*. Birkhäuser, Boston, 1994.

[16] G. H. Golub and C. F. van Loan *Matrix computations*. Johns Hopkins University Press, Baltimore, Maryland, 1996.

[17] C. M. Hoffmann. The problem of accuracy and robustness in geometric computation. Report CSD-TR-771, Dept. Comput. Sci., Purdue Univ., West Lafayette, IN, 1988.

[18] C.M. Hoffmann. How solid is solid modeling? In *Applied Computational Geometry*, volume 1148 of *LNCS*, pages 1–8. Springer, 1996.

[19] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick. Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl.*, 9(6):50–59, November 1989.

[20] C.Y. Hung, B. Parhami. An approximate sign detection method for residue numbers and its application to RNS division. *Computers Math. Applic.* 27(4):23–35, 1994.

[21] J. Keyser, S. Krishnan, and D. Manocha. Efficient B-rep generation of low degree sculptured solids using exact arithmetic. Technical Report 40, Dept. Computer Science, Univ. N. Carolina, Chapel Hill, 1996.

[22] D.E. Knuth. *The Art of Computer Programming: Seminumerical Algorithms*, volume 2. Addison-Wesley, Reading, Massachusetts, 1981 and 1997.

[23] M. Lauer. Computing by homomorphic images. In B. Buchberger, G.E. Collins, and R. Loos, editors, *Computer Algebra: Symbolic and Algebraic Computation*, pages 139–168. Springer, Wien, 2nd edition, 1982.

[24] M.C. Lin, D. Manocha, and J. Canny. Efficient contact determination for dynamic environments. In *Proc. IEEE Conf. Robotics and Automation*, pages 602–608, 1994.

[25] V. Y. Pan, Y. Yu, and C. Stewart. Algebraic and numerical techniques for the computation of matrix determinants. *Computers & Math. (with Applications)*, 1997, to appear.

[26] V. Y. Pan. Parallel computation of polynomial GCD and some related parallel computations over abstract fields. *Theoretical Computer Science*, 162:2, 173–223, 1996.

[27] A. Rege. A complete and practical algorithm for geometric theorem proving. In *Proc. ACM Symp. on Computational Geometry*, pages 277–286, Vancouver, June 1995.

[28] J.R. Shewchuk. Robust adaptive floating-point geometric predicates. In *Proc. 12th Annu. ACM Sympos. Comput. Geom.*, pages 141–150, 1996.

[29] K. Sugihara and M. Iri. A robust topology-oriented incremental algorithm for Voronoi diagrams. *Internat. J. Comput. Geom. Appl.*, 4:179–228, 1994.

[30] J. Wiegley, A. Rao, and K. Goldberg. Computing a statistical distribution of stable poses for a polyhedron. In *Proc. 30th Annual Allerton Conf. on Comm. Control and Computing*, Univ.Ill. Urbana-Champaign, 1992.

[31] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7:3–23, 1997.

[32] C. K. Yap. Exact computational geometry and tolerancing metrology. In D. Avis and J. Bose, editors, *Snapshots of Computational and Discrete Geometry, Vol. 3, Tech. Rep. SOCS-94.50*. McGill School of Comp. Sci., 1995.

[33] C. K. Yap and T. Dubhe. The exact computation paradigm. In D. Du and F. Hwang, editors, *Computing in Euclidean Geometry*. World Scientific Press, 1995.